ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа физико-математических и компьютерных наук

Филиппов Денис Дмитриевич

РАЗРАБОТКА КЛИЕНТСКОГО ПРИЛОЖЕНИЯ СЕРВИСА ЯНДЕКС.ЗАПРАВКИ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКА FLUTTER

Выпускная квалификационная работа – БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
А.А. Прокофьев

Руководитель
канд.физ.-мат.наук, доцент
М.С. Мухин

Санкт-Петербург 2023

Оглавление

Аннотация **3**

Введение **5**

1. Разработка архитектуры **8**

1.1. Введение в разработку на Flutter 8

1.2. Обзор разработанной архитектуры 14

1.3. Выводы 17

2. Экран выбора способа оплаты **18**

2.1 Общее устройство экрана 18

2.2 Сложности реализации интерфейса 23

2.3 Выводы 26

3. Процесс оплаты заказа **27**

3.1 Общее устройство 27

3.2 Особенности реализации диалоговых шторок 31

3.3 Выводы 33

Заключение **35**

Список литературы **36**

Приложение **37**

Аннотация

Сервис Яндекс.Заправки позволяет осуществлять заправку через мобильное приложение, не выходя из машины. Клиентское приложение сервиса разрабатывается отдельно для каждой из поддерживаемых платформ. Для оптимизации процесса разработки было принято решение начать разработку клиентского приложения с помощью фреймворка Flutter, позволяющего реализовывать кроссплатформенные приложения.

Целью работы стала реализация системы оплаты заправки в мобильном приложении с помощью фреймворка Flutter. В ходе работы была изучены основные принципы разработки Flutter приложений, а также разработана архитектура Flutter приложения, на основе которой в дальнейшем реализовывались различные компоненты системы оплаты. В процессе реализации компонент системы оплаты был выявлен ряд ограничений фреймворка и найдены различные пути их преодоления.

В результате была оптимизирована дальнейшая поддержка и развитие системы оплаты заказа, а также создана основа для дальнейшего развития компонент клиентского приложения на Flutter.

Ключевые слова: Яндекс.Заправки, мобильная разработка, кроссплатформенная разработка, фреймворк Flutter

Yandex.GasStations is a service that allows refueling at a gas station using a mobile application without having to get out of the car. The client application of the service is developed for each of the target platforms independently. Due to this, it was decided to start developing a client application using the Flutter framework, which allows implementing cross-platform applications, to optimize the development process.

The purpose of the work was to implement a refueling payment system in a mobile application using framework Flutter. In the course of the work, the basic principles and approaches of Flutter application development were studied and the architecture, based on which various components of the payment system were later implemented, was developed.

In the process of implementing the components of the payment system several framework limitations were identified and various ways to overcome them were found.

As a result, the further support and development of the order payment component was optimized, and the basis for the further component of the client application on Flutter was developed.

Keywords: Yandex.GasStation, mobile development, cross-platform development, Flutter framework

Введение

Яндекс.Заправки – это сервис, позволяющий осуществлять заправку на АЗС с помощью мобильного приложения без необходимости выходить из машины [1]. Для этого пользователь, подъехавший к АЗС, выбирает эту АЗС в приложении, указывает топливо и его количество, оплачивает заправку и после этого заправщик или сам пользователь заливает топливо в бак.

Клиентские приложения разрабатываются для двух платформ – операционных системах Android и IOS. Также существует веб-версия приложения, обладающая несколько ограниченным функционалом.

Каждое из клиентских приложений Яндекс.Заправок разрабатывается нативно, т.е. с использованием специальных технологий разработки для каждой из поддерживаемых платформ. Поэтому для разработки каждого из них требуются отдельные временные ресурсы, вычислительные ресурсы, своя команда разработки, а также отдельное внимание на согласования функций, внедряемых в каждое из клиентских приложений.

При разработке новой функции необходимо отдельно для каждой платформы заниматься:

- реализацией,
- тестированием,
- и дальнейшей поддержкой этой функции

Это накладывает ограничения на скорость внедрения новых функций в приложение.

Также некоторые части кодовой базы собираются в специальный SDK, который используется для встраивания Яндекс.Заправок в другие сервисы, например Яндекс.Навигатор, Яндекс.Карты и др. Для каждой из

поддерживаемых платформ приходится собирать свою версию SDK, что также замедляет процесс его обновления.

В связи с этим для оптимизации процесса разработки было принято решение начать разработку клиентского приложения с помощью кроссплатформенных технологий. В качестве технологии кроссплатформенной разработки был выбран фреймворк Flutter, предназначенный для языка программирования Dart.

Dart – это Си-подобный язык программирование, обладающий статической типизацией и null-безопасностью [2]. Он в достаточной степени похож на такие языки как Java, Kotlin, Swift, использующиеся в мобильной разработке, также обладает сборщиком мусора, интерфейсами, расширениями и другими особенностями. При этом Dart является однопоточным языком программирования с поддержкой асинхронности.

Flutter – это фреймворк для создания мобильных приложений под Android и IOS, веб-приложений, а также десктопных приложений [3]. Он позволяет реализовывать пользовательский интерфейс сразу для всех платформ с помощью единой кодовой базы, а также использовать все доступные ресурсы каждой из целевых платформ.

Цель и задачи работы

Целью данной работы стала реализация системы оплаты заправки в мобильном приложении с помощью фреймворка Flutter.

Задачи:

- разработать базовую архитектуру Flutter приложения Яндекс.Заправок
- реализация экрана способов оплаты

- реализация экранов процесса оплаты
- сравнение нативной реализации системы оплаты с реализаций на Flutter

Структура работы

В главе 1 описан процесс разработки приложений на Flutter, представлена разработанная архитектура, произведено сравнение разных архитектурных подходов к построению Flutter приложения и дано обоснование выбора одного из них.

Глава 2 посвящена реализации экрана способов оплаты.

Последняя глава посвящена реализации экранов процесса оплаты.

1. Разработка архитектуры

1.1. Введение в разработку на Flutter

Как и во многих языках программирование, программа на языке Dart начинается с запуска функции `main`. Flutter приложение представляет собой виджет, отображаемый на экране приложения. Виджеты бывают простыми – например, виджет `Text`, отображающей какой-то текст, на экране – и составными, т.е. содержащими в себе другие виджеты, в результате чего образуется дерево виджетов. Пример простого приложения на Flutter, отображающего на экране текст «Hello World»:

Листинг 1.1. Пример простого приложения на Flutter

```
void main() {
  runApp(Text(‘Hello World!'));
}
```

Для создание собственного виджета необходимо создать класс, наследуемый от класса `StatelessWidget` или класса `StatefullWidget`. Для создания наследников обоих классов необходимо переопределить метод `build`, который вызывается при построении или перестроении пользовательского интерфейса.

`StatelessWidget` предназначен для создание виджетов, не обладающих собственным состоянием, т.е. набором свойств, изменяющихся в процессе работы приложения. Построение виджетов этого типа происходит только при первом построении интерфейса, а также при перестроении виджета-родителя.

Листинг 1.2. Пример виджета без состояния

```
class HelpNearbyWidget extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) => OtherWidget(...);
}
```

`StatefullWidget` предназначен для создания виджетов, обладающих собственным состоянием. Перестроении таких виджетов происходит в тех же случаях, что и перестроение виджетов без состояния, а также при изменении состояния виджета. Изменение состояние происходит при вызове метода `setState`, принимающем функцию, изменяющую состояния виджета.

Листинг 1.3. Пример виджета с состоянием

```
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}
class _CounterState extends State<Counter> {
  int value = 0;
  @override
  Widget build(BuildContext context) =>
    Button(
      child: Text("Value: $value"),
      onPressed: () {
       setState(() => value++);
      },
    );
}
```

Зачастую виджеты во всем дереве обладают общими параметрами, либо при создании одного составного виджета ему в конструктор передаются параметры, которые нужны лишь одному дочернему виджеты. Чтобы не передавать через конструкторы эти параметр существует специальный тип

виджетов – унаследованные виджеты. Это виджеты, которые не отображаются на экране, находятся в дереве виджетов и доступ к полям которых есть у всех дочерних виджетов.

Листинг 1.4. Пример использования унаследованного виджета для получения цвета фона текущий темы приложения

```
class ThemeInheritedWidget extends InheritedWidget {
  ColorScheme scheme;
  ThemeInheritedWidget({this.scheme, super.child});
}
// При построении родительского виджета:
ThemeInheritedWidget(
    child: Page(
        child: MyWidget(...)
        )
    )
)
// При построении дочернего виджета:
ThemeInheritedWidget inherited = ThemeInheritedWidget.of(context);
return Container(color: inherited.scheme.backgroundColor);
```

Экраны Flutter приложения представляют собой виджеты. Эти виджеты хранятся в специальном стеке. Каждый экран, находящийся выше по стеку, отображается поверх нижних экранов, полностью или частично перекрывая их.

В стек экранов добавляется на сам виджет экрана, а контейнер `Route`. Экземпляр класса `Route` обладает именем экрана, конструктором виджета данного экрана, а также аргументами – данные любого типа, которые впоследствии могут быть использованы для построения виджета.

Управление стеком экранов происходит через встроенный виджет `Navigator`. Это унаследованный виджет, к которому имеют доступ все дочерние виджеты. Из любого унаследованного виджета можно положить новый `Route` в стек, получив экземпляр класса `Navigator` и вызвав метод `push`.

По умолчанию у приложения есть корневой `Navigator` со своим стеком. Но при необходимости можно создать свой экземпляр класса `Navigator` со своим стеком. Это может понадобиться в случаях, когда экраны стека должны отображаться не на весь экран устройства, а лишь на каком-то ограниченном пространстве (например, в диалоговой шторке). При создании своего экземпляра Navigator можно переопределить метод `onGenerateRoute`, создающий по названию экрана определенный `Route`, который далее добавляется в стек. А далее использовать метод `pushNamed`, который добавит в стек экран с определенным названием.

Листинг 1.5. Пример использования виджет Navigator для навигации между экранами

```
// в родительском виджете
Widget build(BuildContext context) =>
  Navigator(
    onGenerateRoute: (settings) {
      switch (settings.name) {
        case "PageA":
          return Route(builder: () => MyWidgetA(settings.arguments));
        case "PageB":
          return Route(builder: () => MyWidgetB(settings.arguments));
      }
    },
```

```
    initialRoute: "PageA”,
  );
// где-то в дочернем виджете
Widget build(BuildContext context) =>
    Button(
      child: Text("Go to Page B"),
      onPressed: () {
        Navigator.of(context).pushNamed("PageB");
      },
    );
```

На основе вышеизложенного уже можно реализовать довольно сложные приложения со сложной бизнес-логикой. В таком случае вся бизнес-логика приложения будет реализована внутри виджетов. Но виджет должен отвечать только за отображение своего состояния на экране, поведение приложения должно задаваться в отдельной компоненте.

В качестве такой компоненты в данной работе используется шаблон «Компонента бизнес-логики» (Business Logic Component – BLoC), широко распространенный в разработке на Flutter. Суть шаблона заключается в следующем [4]:

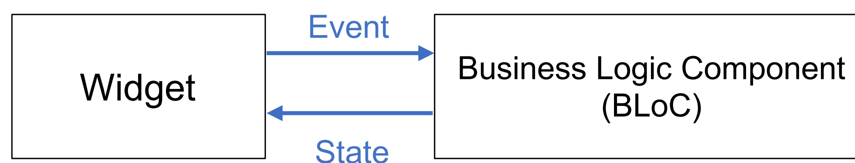

Рис. 1.1. Схема шаблона Компонента бизнес-логики

При наступлении какого-то события (нажатие на кнопку, сворачивание экрана и т.п.) информация о нём отправляется в компоненту бизнес-логики. Компонента обрабатывает это событие и в ответ отправляет виджету его новое

состояние (например, новые данные для отображения), в соответствии с которым виджет перестраивается.

Переход между экранами, осуществляющийся с помощью виджета `Navigator`, является частью бизнес-логики, поэтому должен происходить внутри BLoC. Но при этом реализация перехода между экранами не должна влиять на реализацию компоненты бизнес-логики. К тому же переключение экранов может осуществляться как через прямое обращение к `Navigator`, так и с помощью сторонних библиотек.

В связи с вышеуказанным, навигацию между экранами принято выносить в отдельный интерфейс, реализация которого в дальнейшем используется внутри компоненты бизнес-логики.

Взаимодействие с сервером можно осуществлять посредством обычного http-клиента. Получение и отправка данных также является частью бизнес-логики, поэтому должно быть инициировано там же. При этом компоненте бизнес-логике не нужно знать о том, как именно происходит работа с данными.

Из-за этого, а также для удобства тестирование, взаимодействие с сервером выносится в отдельный интерфейс

Зачастую приложения на Flutter хоть и являются кроссплатформенными, всё же обладают компонентами, которые реализованы нативно. Связано это может быть, например, необходимостью использовать сторонние библиотеки, которые обладают только нативной реализацией.

Для обмена данными между нативной частью приложения и частью на Flutter существует канал взаимодействия с нативной частью приложения – класс `MethodChannel`. Этот канал создается как в нативной, так и в кроссплатформенной частях приложения. Через этот канал можно вызывать различные методы, например открытие какого-то экрана или получение данных.

Т.к. вызов нативных компонент тоже является частью бизнес-логики, `MethodChannel` стоит использовать внутри `BLoC`.

1.2. Обзор разработанной архитектуры

Базовая архитектур, на основе которой будут реализованы различные компоненты клиентского приложения, выглядит следующим образом:

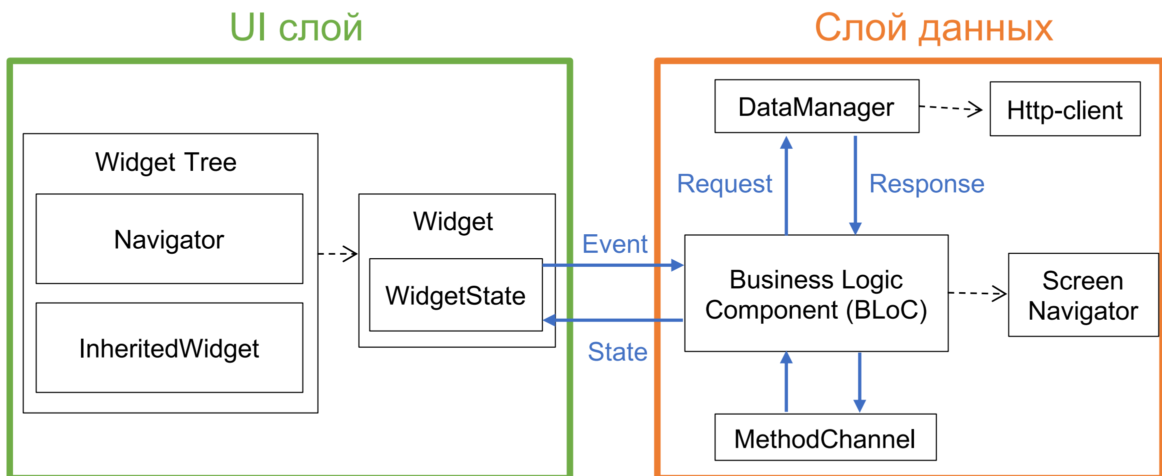

Рис. 1.2. Схема разработанной архитектуры Flutter приложения

При построении пользовательского интерфейса создается дерево виджетов. Часть виджетов, обладающих сложной бизнес-логикой, обладает выделенной для данного виджета компонентой бизнес-логики. Компонента обладает рядом интерфейсов, позволяющих эту логику осуществлять

(интерфейс управления данным, навигации между экраном, канал взаимодействия с нативной частью приложения).

Получившуюся схему можно разделить на 2 слоя: слой пользовательского интерфейса и слой данных. Это разделение является общепринятым в мобильной разработке. Вопрос, как именно это разделение произвести, является ключевым при разработке архитектуры.

В данной работе для разделения на слои выбран шаблон компонента бизнес-логики.

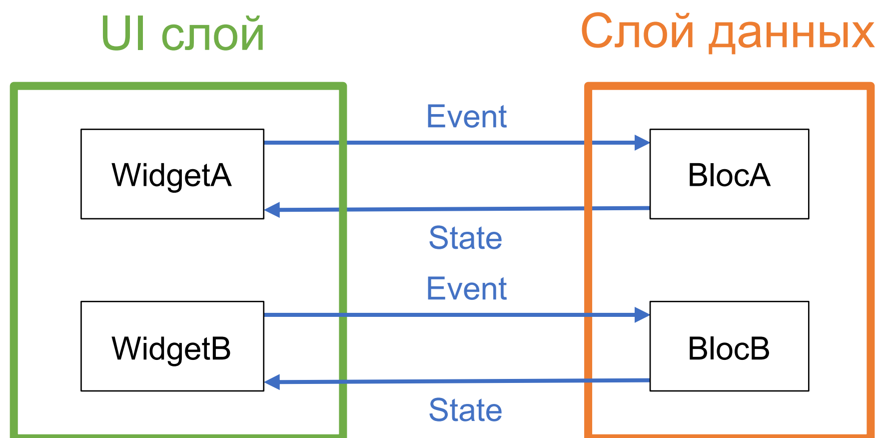

Рис. 1.3. Схема шаблона «Компонента бизнес-логики»

Этот шаблон предоставляет однозначное и легко реализуемое разделение на слои, не зависящие друг от друга. При этом каждый сложный виджет обладает своей компонентой бизнес-логики, что дает возможность разрабатывать и использовать различные компоненты приложения независимо друг от друга, что в будущем будет крайне полезно при переиспользовании и выносе части компонент приложения в SDK.

Существуют и ряд других подходов для разделения архитектуры на слои. Рассмотрим шаблон Redux [5]:

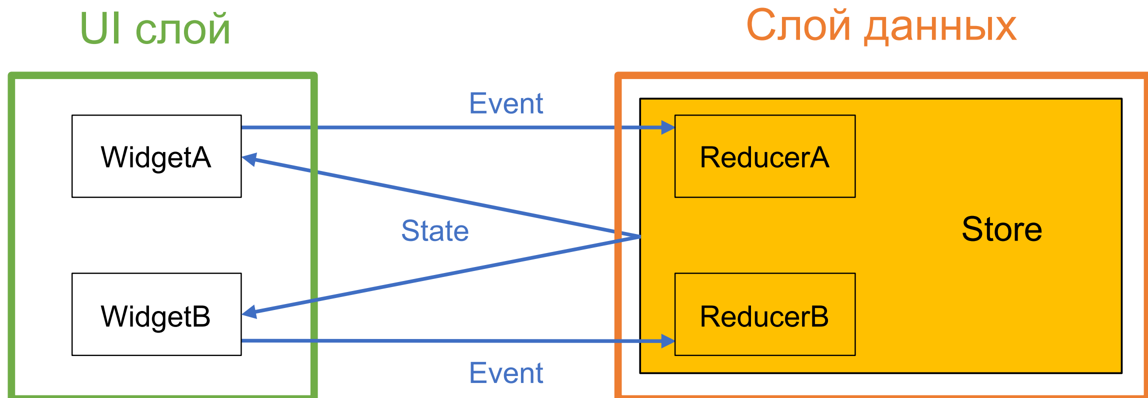

Рис. 1.4. Схема шаблона Redux

В отличии от шаблона «Компонента бизнес-логики», каждому сложному виджету соответствует своя компонента бизнес-логики, в шаблоне Redux присутствует одна структура, называемая хранилищем, внутри которой реализуется вся бизнес-логика. При наступлении пользовательского события виджет отправляет в хранилище информацию об этом событии. Это событие обрабатывается в соответствующем событию обработчиком. Также виджеты подписываются на обновление определенных данных из хранилища и таким образом получают новые данные для отображения.

Этот шаблон подходит для тех приложений, чье состояния обновляется достаточно часто по независящим от пользователей причинам и при этом многие виджеты зависит от одних и тех же данных. При этом Redux не позволяет выносить отдельные компоненты в SDK, т.к. в таком случае вместе с этими компонентами необходимо сохранять в SDK всё хранилище.

Другие шаблоны так же в отличии шаблона «Компонента бизнес-логики» обладают рядом недостатков. Среди них:

- смешение слоев пользовательского интерфейса и слоя данных

- сложность в переиспользовании компонент и вынос их в SDK

- сложность реализации по сравнению с шаблоном «Компонента бизнес-логики»

1.3. Выводы

В данной главе были рассмотрены основы разработки Flutter приложений, а также спроектирована базовая архитектура Flutter приложения, в соответствии с которой в дальнейшем будут разрабатываться компоненты приложения.

В качестве архитектурного шаблона, на основе которого была разработана архитектура, была выбрана «Компонента бизнес-логики», в силу удобства для разделения слоев пользовательского интерфейса и данных, а также удобства выноса разработанных компонент приложения в SDK.

2. Экран выбора способа оплаты

2.1 Общее устройство экрана

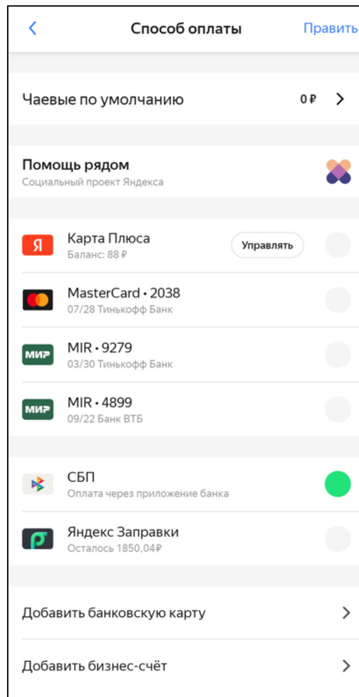

Рис. 2.1. Интерфейс экрана выбора способа оплаты

Экран выбора способа оплаты предоставляет возможность:

- выбрать способ оплаты
- добавить новый или удалить ранее добавленный способ оплаты
- добавить бизнес-счет
- задать значение чаевых, которые будут отправлены заправщику, по умолчанию
- стать участником благотворительного проекта «Помощь рядом»
- управлять счетом или завести счет в Яндекс Банке

Архитектура экрана почти полностью совпадает с базовой архитектурой, описанной в предыдущей главе (для обозначения элементов, относящихся к рассматриваемому экрану, используется префикс `Wallet`):

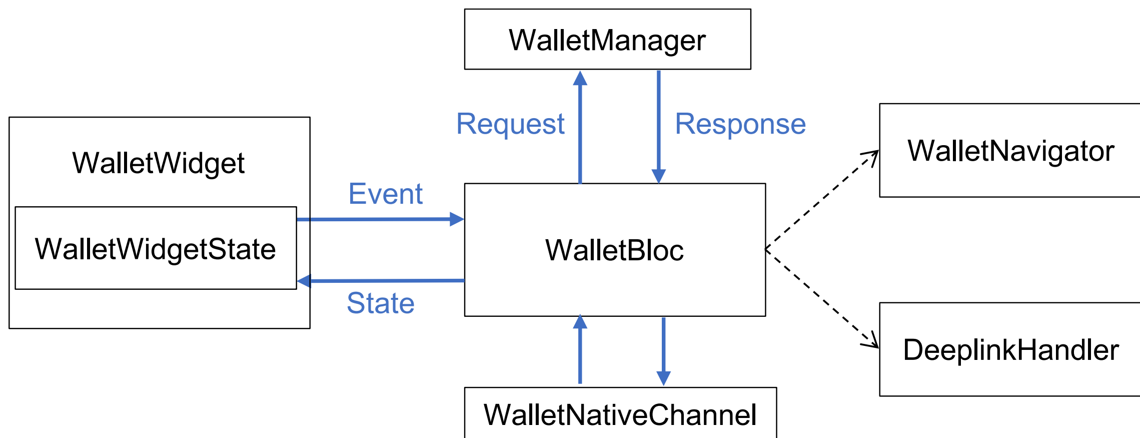

Рис. 2.2. Архитектура экрана выбора способа оплаты

Интерфейс экрана представлен виджетом с состоянием `WalletWidget`, бизнес-логика реализуется в компоненте `WalletBloc`. Отличие от базовой архитектуру заключается в том, что для реализация работу некоторых элементов интерфейса был реализован `DeeplinkHandler` – обработчик глубинных ссылок, которые используется для открытия определенных экранов внутри приложения.

Виджет экрана состоит из двух основных составляющих: панель навигации и тело экрана. Содержание виджета зависит от присланного компонентой бизнес-логики состояния.

Всего есть три возможных состояния:

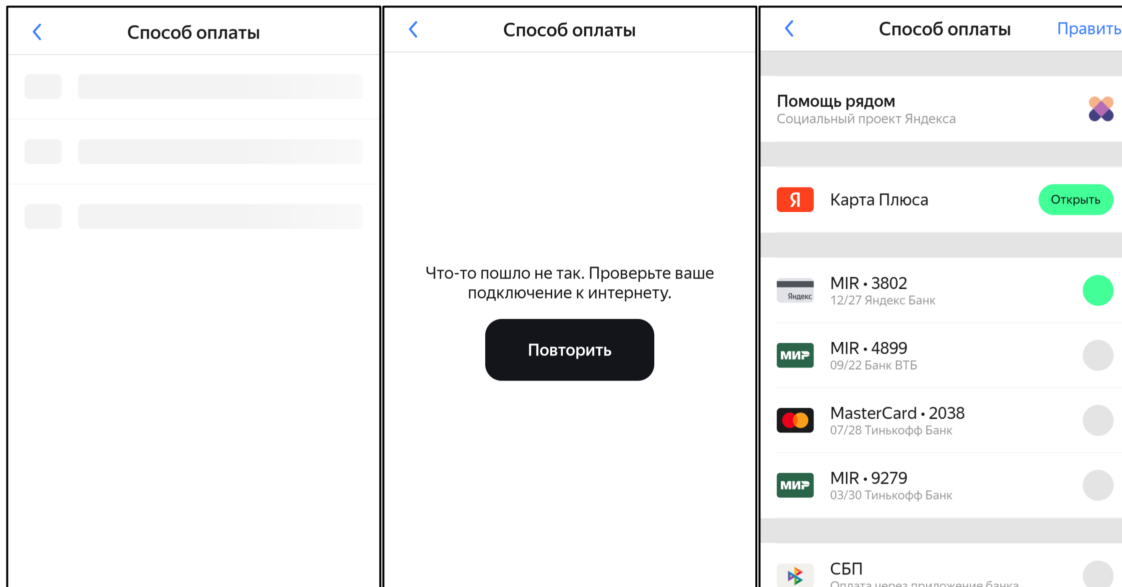

Рис. 2.3. Состояния экрана выбора способа оплаты

1) Загрузки. Приходит в процессе загрузки данных. В случае этого состояния `WalletWidget` отображает список мерцающих виджетов. Эффект мерцания реализован с помощью внешней библиотеки.
2) Ошибки загрузки данных. Пользователю отображается кнопка с предложением попробовать повторить загрузку, при нажатии на которую экран снова переходит в состояние загрузки.
3) Данные успешно загружены. Модель состояния, которое отправляет компонента бизнес-логики виджету в данном случае, выглядит так:

Листинг 2.1. Модель виджета состояния успешной загрузки данных

```
class WalletWidgetState {
  final List<ListItemModel> models; // модели виджетов, которые
отображаются на экране
  final bool blockScreen; // заблокирован ли экран
  final bool editMode; // включен ли режим редактирования
  final String? editModeStr; // какой текст отображать на кнопке
редактирования
  final int selectedPaymentIndex; // каким в списке моделей идет
```

```
модель выбранного способа оплаты (нужен для начального прокручивания
списка до текущего способа оплаты)
}
```

Тело экрана представляет собой виджет `ListView`, состоящий из виджетов, каждый из которых соответствует элементу `WalletWidgetState.models`.

Основным элементом экрана является виджет, отображающий способ оплаты. При нажатии на способ оплаты в режиме выбора компонента бизнес-логики отправляет через `WalletManager` информацию о выбранном способе оплаты на сервер, и пока не придёт ответ об успешном выборе, экран не обрабатывает пользовательский ввод. В режиме редактирования способ оплаты, который может быть удален, удаляется локально, без ожидания ответа от сервера, и экран не блокируется.

Также есть особый способ оплаты – карта Яндекс Банка. Помимо выбора данного способа оплаты, на экране есть возможность открыть личный кабинет или завести счет в Яндекс Банке, нажав на специальную кнопку, открывающую приложение Яндекс Банка через специальный плагин. Плагин на Flutter – пакет, предоставляющий различный функционал, реализованный нативно под каждую поддерживаемую платформу. Подключение данного плагина оказалось не тривиальной задачей, т.к. потребовалось изменить версии других используемых пакетов из-за конфликта версий.

Для добавления новых способов оплаты был реализован механизм действий. Сервер присылал список действий, содержащий название действия и ссылку. Ссылка может быть как у глубинной, так и ссылкой веб-страницу. Для обработки возможных действий реализован обработчик глубинных

ссылок `DeeplinkHandler`. Если обработчик смог обработать ссылку как глубинную, запускается соответствующий экран, иначе ссылка открывается через `WebView` – виджет, позволяющий открывать веб-страницы.

На данный момент возможных способов оплаты два.

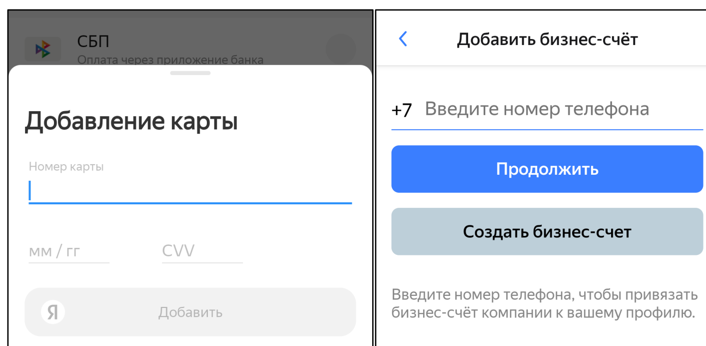

Рис. 2.4. Интерфейс добавления способа оплаты

- добавление карты, осуществляется с помощью внутреннего SDK Яндекса, подключаемого через плагин.
- добавление бизнес-счета, осуществляется через `WebView`.

Также на экране есть возможность задать значение чаевых, оставляемых заправщику, по умолчанию. Исходно это значение хранилось локально на устройстве, в новом экране оно присылается с сервера. Но для сохранения обратной совместимости по каналу взаимодействия с нативной частью `WalletNativeChannel` происходит запрос значения из нативной части приложения.

Последняя функция на экране выбора способа оплаты – возможность поучаствовать в благотворительном проекте «Помощь рядом». При нажатии на виджет открывается диалоговая шторка, внутри которой запускается `WebView` со страницей благотворительного проекта. Подробнее о реализации будет рассказано в следующей главе.

2.2 Сложности реализации интерфейса

При реализации интерфейса экрана выбора способа оплаты возникло несколько проблем.

Первой стало отображение в диалоговой шторке страницы «Помощи рядом». При прокручивании веб-страницы до верха дальнейшие попытки прокрутки должны были привести к сворачиванию диалогового окна. Но этого не происходило.

Связано это было с тем, что `WebView` реализован через плагин, следовательно на каждой платформе отображение веб-страницы реализовано нативно. Из-за этого Flutter не знает о том, насколько была прокручена веб-страница.

В качестве временного решения поверх WebView в верхней части диалоговой шторки отображается невидимый контейнер. Таким образом пользователь сможет потянуть за верхнюю часть шторки и тем самым скрыть её:

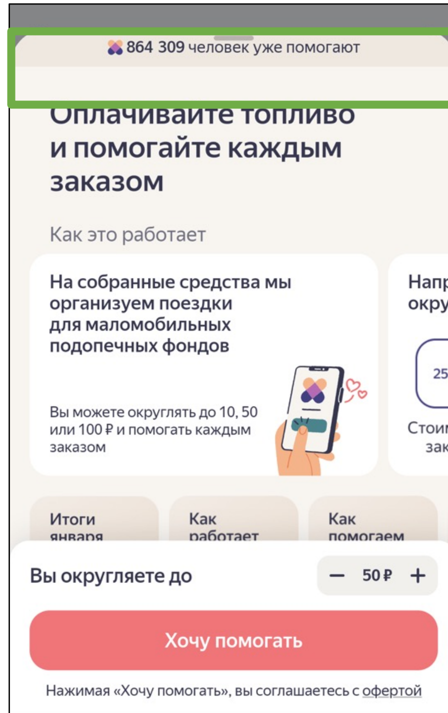

Рис. 2.5. Решение проблемы сворачивания диалоговой шторки с WebView

Более основательных решений проблемы передачи данных о прокрутке веб-страницы может быть два:

- передача данных о прокрутке по нативному каналу взаимодействия, что требует доработки стандартного `WebView`.
- реализация веб-движка непосредственно через инструменты Flutter, что является очень объемной задачей.

В будущем будет произведена попытка реализовать первое решение.

Второй проблемой стала реализация следующей функциональности. При обновлении экрана выбора способа оплаты необходимо было прокрутить список до текущего способа оплаты, если он не помещается на экране:

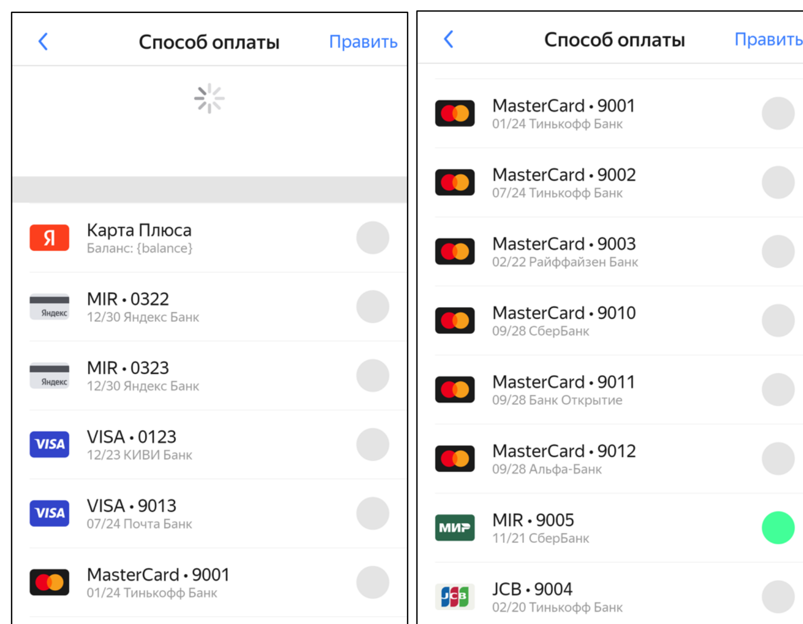

Рис. 2.6. Обновление экрана и прокрутка до текущего способа оплаты

У этой задачи есть готовые решения. Во-первых, можно виджету выбранного способа оплаты задать определенный ключ (играющий роль идентификатора) и передать этот ключ в метод `Scrollable.ensureVisible`, который автоматически докрутит список, если виджет с данным ключом на экране не отображен. Во-вторых, есть разные внешние библиотеки, которые также позволяют прокрутить список до определенного элемента.

При этом требовалось использовать индикатор обновления определенного вида. Есть два способа этого добиться:

- реализовать индикатор, что является достаточно трудоемкой задачей
- использовать готовую стандартную реализацию `CupertinoSliverRefreshController`.

Проблема стандартной реализации заключается в том, что она совместима лишь с виджетами пакета `Sliver`, предназначенного для реализации эффектов, связанных с прокруткой [6]. Но для виджетов этого пакета нет готовых решений задачи прокрутки. Причиной является достаточно

серьезная оптимизация виджетов данного пакета, из-за которой виджеты, находящиеся далеко от экрана и до которых список прокручивать далеко, не создаются. Им соответственно не присваивается ключ и использовать метод `Scrollable.ensureVisible` не удается. Сторонних пакетов, решающих данную задачу, также не найдено.

Сначала была произведена попытка доработать реализацию виджета `SliverList`, но реализация данного виджета содержит большое количество инвариантов, изучение и соблюдение которых достаточно трудоемкая задача.

В итоге было найдено более простое решение – использовать `SliverList`, а в него положить обычный виджет `Column`, содержащий список способов оплаты. Это, позволило воспользоваться стандартным решением задачи прокручивания, т.к. виджет Column осуществляет построение сразу всех дочерних виджетов. Но это сводило к нулю оптимизации виджетов из `Sliver` пакета. В данном случае это оправдано, т.к. обычно способов оплаты немного, и оптимизациями можно пренебречь. Но в будущем можно будет найти более оптимальное решение задачи прокрутки, т.к. оно может понадобиться при реализации других компонент приложения.

2.3 Выводы

В данной главе были описано общее устройство экрана выбора способа оплаты, описаны некоторые детали реализации, а также основные проблемы, с которыми пришлось столкнуться в процессе разработки.

В результате был реализованы все функции экрана выбора способа оплаты, а также намечены пути улучшения текущей реализации.

3. Процесс оплаты заказа

3.1 Общее устройство

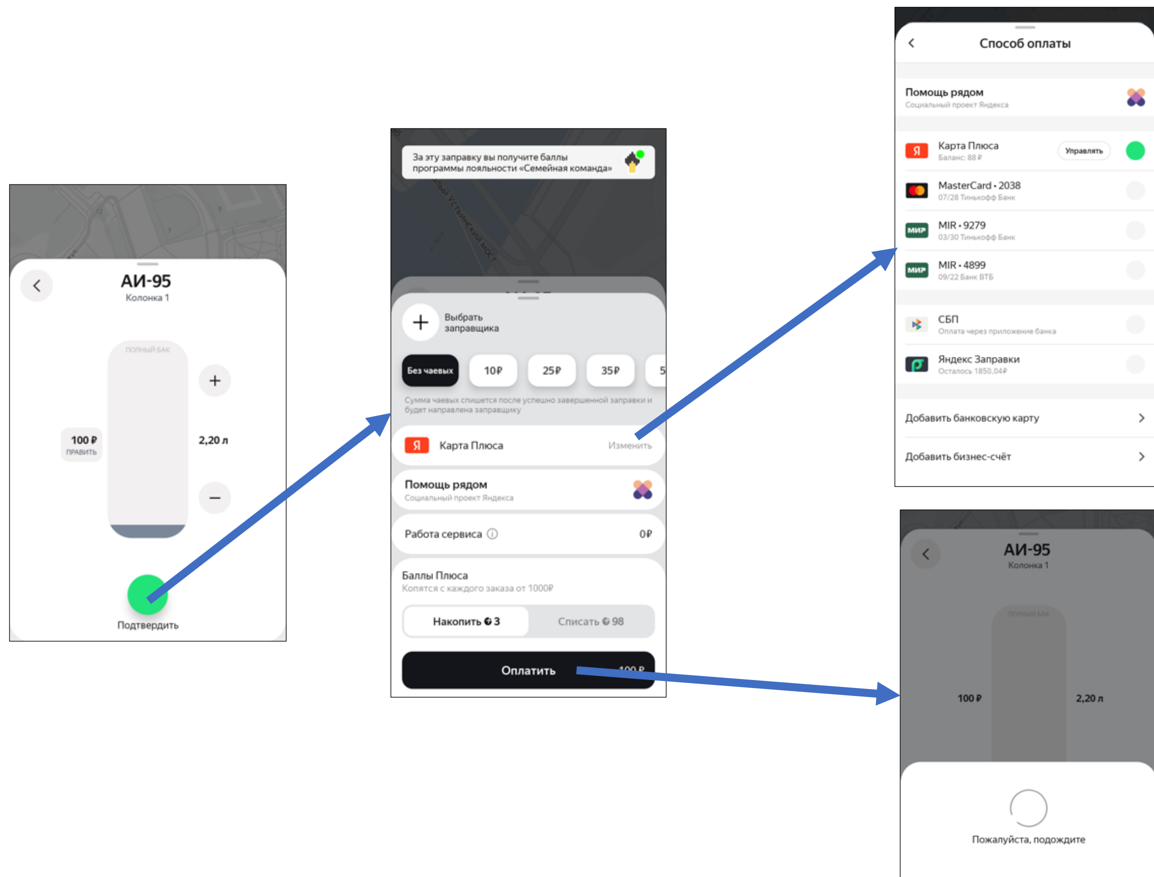

Рис. 3.1. Процесс оплаты заказа

Процесс оплаты заправки в приложении устроен следующем. Пользователю открывается диалоговая шторка, отображающая различные параметры оплаты заказа. В том числе пользователь может выбрать способ оплаты заказа, для выбора которого откроется диалоговая шторка с экраном, описанным в предыдущей главе. После нажатия на кнопку оплатить, шторка с параметрами оплаты сворачивается и открывается диалоговая шторка, отображающая статус оплаты заказа.

Архитектуры экрана выглядит следующим образом (для обозначения элементов, относящихся к рассматриваемому экрану, используется префикс `Payment`):

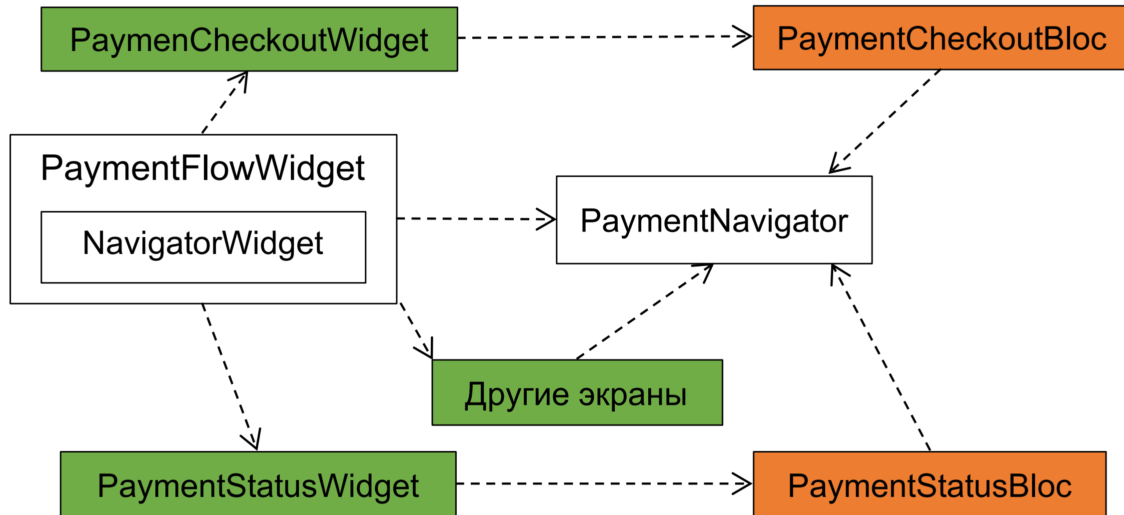

Рис. 3.2. Архитектура экрана процесса оплаты заказа

Корневым виджетом дерева является `PaymentFlowWidget`, он занимает весь экран устройства и внутри него происходит переключения между экранами (диалоговыми шторками), для чего одним из детей в дереве является `NavigatorWidget`.. Основные экраны представлены виджетами `PaymentCheckoutWidget` – экран параметров оплаты – и `PaymentStatusWidget` -- экран статуса оплаты, – каждый из которых обладает своей компонентой бизнес-логики. Переключением между экранами выделено в отдельный интерфейс `PaymentNavigator`, которым пользуется как `PaymentFlowWidget`, определяющий начальный экран (им может быть и экран статуса оплаты, если приложение было перезапущено после нажатия на кнопку оплаты), так и все компоненты бизнес-логики. Также могут запускаться и другие экраны, например экран выбора способа оплаты, который для навигации также использует `PaymentNavigator`.

Как и у экрана выбора способов оплаты, экран параметров оплаты обладает тремя состояниями: загрузки, ошибки и состояние успешно загруженных данных.

В состоянии загрузки происходит загрузка данных об оплате с учетом выбранных пользователем параметров заказа, а на экране отображаются мерцающие элементы.

В состоянии ошибки также отображается информацию об ошибке и кнопка с предложением попробовать повторить загрузку данных.

В состояние успешной загрузки данных об оплате эти данные отображаются на экране. Изменение любых параметров оплаты всегда влечет за собой повторное получение данных, т.к. стоимость заказа при разных параметрах оплаты может быть разной. При этом экран может как перейти в состояние загрузки – например, при изменении способа оплаты, – так и не перейти – например, когда пользователь изменил списать баллы Яндекс.Плюса, в таком случае будет лишь заблокирована кнопка оплаты до момента, пока не придут новые данные о стоимости заказа.

Успешное состояние задается уже не просто списком моделей виджетов, как это было на экране способов оплаты, а отдельными полями, каждое из которых отвечает за определенный виджет:

Листинг 3.1. Модель успешного состояния экрана параметров оплаты

```
class PaymentCheckoutWidgetState {
  final DiscountsBannerModel? discountModel;
  final PaymentListItemModel selectedPayment;
  final PlusPointsWidgetModel? plusPoints;
  final PayButtonModel payButtonModel;
}
```

Переход от списка моделей к отдельным полям необходим, т.к. некоторые виджеты являются обязательными – способ оплаты и кнопка оплаты всегда будут отображаться, – а некоторые нет – например баллы Яндекс.Плюса доступны не всем пользователям.

Экран статуса оплаты обладает тремя типовыми состояниями: оплачено успешно, оплачено не успешно и получение статуса заказа. Каждое состояние отображается в диалоговой шторке с помощью соответствующей иконки и подписи.

Получение статуса оплаты происходит через периодический опрос двух источников: с бекенда, который узнает статус оплаты из платежной системы, и напрямую из платежной системы. Если один из источников указывает, что оплата прошла или случилась ошибка, меняется состояние виджета. Опрос двух источников происходит на случай, если связь с одним из них по какой-то причине прервется и не восстановится.

Если в качестве способа оплаты выбрана «Система быстрых платежей», то с помощью внутреннего SDK Яндекса (того же, через который происходит добавление карты) открывается диалоговая шторка с интерфейсом, внутри которого происходит выбор банка и оплата заказа.

У всех экранов, отображаемых в процессе оплаты заказа, есть довольно много общих компонент:

- `PaymentNavigator`, используемый для переключениями между экранами
- Классы-менеджеры для взаимодействия с бекендом
- Информация о текущем заказе

- и др.

Чтобы все эти зависимости внедрять, не передавая их через конструкторы использован наследуемый виджет, реализующий следующий интерфейс:

Листинг 3.2. Интерфейс для внедрения зависимостей

```
abstract class DataProvider {
  void registerSingleton<T extends Object>(T instance);
  T get<T extends Object>();
}
```

Интерфейс был реализован с помощью библиотеки GetIt [7].

3.2 Особенности реализации диалоговых шторок

Для реализации диалоговых шторок возник ряд трудностей.

Содержимое шторки может не поместиться на экране. Поэтому виджет, содержащийся внутри диалоговой шторки должен быть прокручиваемым.

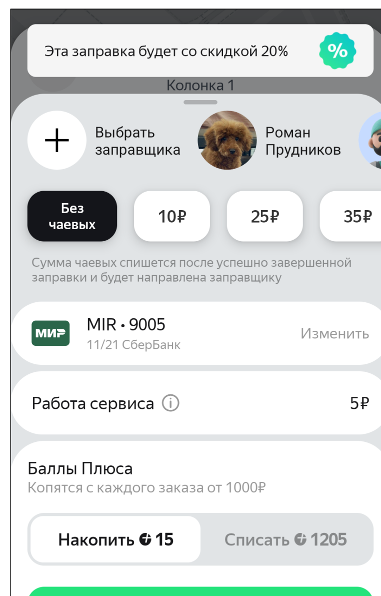

Рис. 3.3. Пример содержимого диалоговой шторки, не поместившегося на экране

Но если в стандартную диалоговую шторку положить виджет, который может прокручиваться, шторка перестанет сворачиваться, т.к. все касания к шторки будут перехватываться внутренним прокручиваемым виджетом. Поэтому, чтобы сохранить возможность и прокручивать содержимое, и сворачивать шторку, необходимо было реализовать шторку со своим обработчиком касаний.

При этом нельзя было взять стороннюю реализацию диалоговой шторки. Над шторкой должен отображаться баннер, в котором может дополнительная информация о заказе. Этот баннер при сворачивании шторки должен постепенно исчезать:

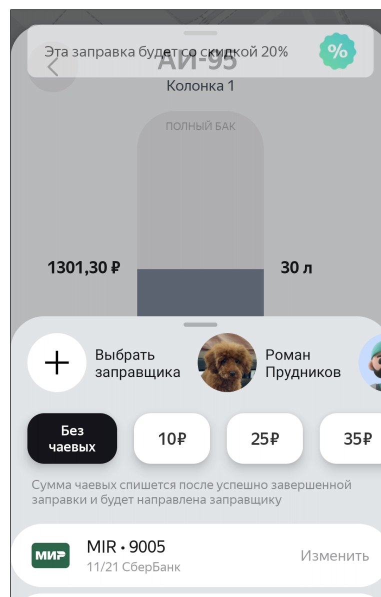

Рис. 3.4. Исчезновение баннера над диалоговой шторкой

И реализовать подобную анимацию можно двумя способами:

1) В реализации диалоговой шторки создать публичные переменные, значение которых указывало бы на её текущее состояние и дальше эти переменные использовать для реализации разных эффектов вне реализации шторки. Но подобный подход оказался плох с

архитектурной точки зрения, как минимум из-за того, что ответственность за обработку нажатия на пустое пространство между шторкой и баннером переносилась бы на пользователя данной шторки.

2) Встроить исчезновение баннера в реализацию диалоговой шторки. Этот подход не обладает недостатками предыдущего решения, поэтому он был реализован.

Также проблемой стало переключение между экранами, содержащими диалоговые шторки. Использовать напрямую те методы, которые предоставляет `NavigatorWidget`, нельзя т.к. они не учитывают то, что при закрытии экрана с диалоговой шторкой её нужно свернуть.

Для решения этой проблемы был реализован менеджер диалоговых шторок `CardBottomManager`, который учитывает данную особенность работы со шторками и который использовался при реализации `PaymentNavigator.`

Этот менеджер устроен так: у него есть стек диалоговых шторок, которые были добавлены в стек экранов приложения через `NavigatorWidget.push`. При закрытии экрана с диалоговой шторкой она убирается со стека менеджера, потом вызывается анимация сворачивания шторки, и только после этого содержащий эту шторку экран удаляется со стека экранов с помощью метода `NavigatorWidget.pop`.

3.3 Выводы

В данной главе были описаны процесс оплаты заказа, а также его архитектурное устройство, приведены детали реализации экрана параметров

заказа и экрана статуса оплаты. Также разобраны основные трудности, возникшие при реализации диалоговых шторок, и их решения.

Результатом стала реализация работоспособной компоненты оплаты заказа, которая в дальнейшем может быть встроена в различные сценарии, доступные в приложении, а также при необходимости расширена дополнительными параметрами оплаты.

Заключение

В данной работе был реализована система оплаты заказа в мобильном приложении Яндекс.Заправки с использованием фреймворка Flutter, использование которого позволяет запускать реализованную систему на поддерживаемых мобильных платформах – операционных системах Android и IOS.

Были рассмотрены разные подходы к построению архитектуры Flutter приложения и выбрана наиболее подходящая для дальнейшей разработки компонент приложения Яндекс Заправок.

При реализации экрана выбора способа оплаты были выявлены некоторые недоработки фреймворка, не позволяющие реализовать не очень сложный интерфейс. Найдены временные решения этих проблем, а также намечены более оптимальные решения, которые будут применены в будущем.

Для реализации процесса оплаты возник ряд трудностей, связанных с корректной работой диалоговых шторок. Для их решения был разработан виджет диалоговой шторки со своим обработчиком касаний, а также менеджер диалоговых шторок, позволяющий осуществлять правильный переход между диалоговыми шторками.

В ходе работы была накоплена база знаний о Flutter разработке и в том числе получены навыки построение архитектуры Flutter приложение, а также реализации его конкретных компонент. Полученный опыт позволит в дальнейшем переносить другие нативные компоненты на Flutter, а также реализовывать новые компоненты и функции с помощью этого фреймворка. В перспективе всё клиентское приложение и весь SDK Яндекс.Заправок будут реализованы на Flutter, что позволит сервису развиваться более активно.

Список литературы

1. Яндекс.Заправки — оплата топлива из машины // Яндекс.Заправки URL: `https://zapravki.yandex.ru/` (дата обращения: 21.05.2023).
2. Dart overview // Dart URL: `https://dart.dev/overview` (дата обращения: 21.05.2023).
3. Flutter documentation // Flutter URL: `https://flutter.dev/` (дата обращения: 21.05.2023).
4. Architecture // Bloc State Management Library URL: `https://bloclibrary.dev/#/architecture` (дата обращения: 21.05.2023).
5. redux // Dart Package URL: `https://pub.dev/packages/redux` (дата обращения: 21.05.2023).
6. Using slivers to achieve fancy scrolling // Flutter URL: `https://docs.flutter.dev/ui/advanced/slivers` (дата обращения: 21.05.2023).
7. get_it // Dart Package URL: `https://pub.dev/packages/get_it` (дата обращения: 21.05.2023).

Приложение

Нативная разработка – разработка программного обеспечения для определенной платформы с использованием технологий предназначенных для данной платформы.

Software Development Kit (SDK) – набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете, позволяющий встраивать в разрабатываемое ПО дополнительную функциональность, реализованную в рамках этого пакета.

Фреймворк – это абстракция, в которой общий код, предоставляющий универсальную функциональность, может быть выборочно переопределен или специализирован пользовательским кодом, предоставляющим конкретную функциональность.

Http-клиент – библиотека передачи данных, используемая на стороне клиента, предназначенная для отправки и получения данных по протоколу HTTP.

Виджет – конфигурация элемента графического интерфейса пользователя.

Глубинная ссылка (deeplink) – это особый вид ссылок, позволяющий направлять пользователя на конкретную страницу в приложении.

Плагин во Flutter – пакет языка Dart, реализация которого обладает вызовами платформенно зависимого кода.

Web View – веб-браузер, встроенный в приложение и предназначенный для отображения веб-страниц.